A Scalable Parallel Algorithm for Multiple Objective Linear Programs

Malgorzata M. Wiecek* Hong Zhang †

Abstract

This paper presents an ADBASE-based parallel algorithm for solving multiple objective linear programs (MOLPs). Job balance, speedup and scalability are of primary interest in evaluating efficiency of the new algorithm. Implementation results on Intel iPSC/2 and Paragon multiprocessors show that the algorithm significantly speeds up the process of solving MOLPs, which is understood as generating all or some efficient extreme points and unbounded efficient edges. The algorithm gives specially good results for large and very large problems. Motivation and justification for solving such large MOLPs are also included.

*Department of Mathematical Sciences, Clemson University, Clemson, SC 29634, U.S.A. (wmalgor@clemson.clemson.edu). Research supported in part by the National Science Foundation under contract DMS-9308605.

†Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665, U.S.A. On leave from Department of Mathematical Sciences, Clemson University, Clemson, SC 29634, (hongsu@math.clemson.edu). Research supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-19480.

1 Introduction

Complex decision problems related to economy, environment, business and engineering are multidimensional and have multiple and conflicting objectives. In the presence of multiple objectives, a decision problem has to be treated in the multicriteria decision making framework. Multiple objective programming is concerned with generating solution sets of multiple objective problems that usually include a large or infinite number of points referred to as efficient solutions. Those efficient points are then the candidates for optimal solutions of the multicriteria decision making problem. Multicriteria problems are therefore naturally well structured to be solved on parallel architectures. It has been already proven that parallel algorithms offer substantial savings in execution time, facilitate solving more complex computational problems, and make real-time response possible for problems that were previously considered as intractable because of their magnitude.

Several studies on the potential of using parallel processing in the field of multicriteria optimization have been already undertaken. Evtushenko et al. [8] recognized that multicriteria optimization deals with one of the most sophisticated aspects of human activity which is to achieve several goals by a single act of decision making. Driven by this idea, they developed DISO, a dialogue system for solving optimization problems, whose one of the main parts is the multicriteria optimization package. They also suggested a possibility of organizing parallel calculations within this package. The study reported by Climaco et al. [6] seems to be the first completed research task in the area of multicriteria optimization and focuses on using parallel processing in interactive multiple objective linear programming. Grauer and Boden [9] discussed opportunities in parallelization for mathematical programming problems and interactive decision support. Ng and Yang [11] proposed to sample the efficient frontier of a multiple objective program by simultaneously solving related single criterion optimization problems developed using the ϵ-constraint approach [5]. A multiple reference point approach to solving multiple objective linear programs (MOLPs) was developed by Costa and Climaco [7] and parallel processors were kept to control the reference points and help the decision maker search for efficient and optimal solutions. Lewandowski [10] reported parallel implementation of selected multicriteria optimization algorithms.

The field of engineering provides various applications of multicriteria optimization, recently also implemented on parallel architectures. Chang [4] proposed a pattern recognition approach for optimization of power systems

in a multiobjective environment.

The research work reported in this paper, as a continuation of preliminary studies reported in [15] and [16], is related to solving an MOLP which is understood as generating all or a subset of efficient solutions of this problem. MOLPs often have a large number of solutions in the form of extreme efficient points (EEPs) and unbounded efficient edges [5], [12]. The process of finding all of them or even a subset of them is very space and time consuming. Speeding up the process can be naturally supported by new algorithms executed on parallel computers.

The availability of sequential algorithms for generating efficient points of multiple objective programs and rising interest in parallel computations motivated to develop a parallel algorithm for MOLPs. The structure of the efficient set of MOLPs turns out to be very helpful in designing a parallel algorithm. Since every efficient extreme point is connected to every other efficient extreme point by a series of efficient edges, the process of finding efficient points can be organized so that subsets of the efficient set can be generated simultaneously. The parallel algorithm proposed in this work is based on ADBASE [12], [13], a well known sequential computer package for solving MOLPs.

The paper is organized as follows. In the next section the software ADBASE is briefly presented with emphasis on several ideas for its parallelization. Some of these ideas are discussed in more detail since they have been tested in the first stage of this research and resulted in a basic parallel algorithm. Section 3 discusses two strategies that significantly improved the efficiency of the basic parallel algorithm, and includes the actual parallel algorithm. The algorithm has been implemented on an Intel iPSC/2 and a Paragon multiprocessors for many MOLPs, focusing on large and extremely large problems. Its efficiency and scalability have been measured and are reported in Section 4. Incentives for dealing with large multiple objective programs are discussed in the same section. Conclusions as well as some directions for further research are given in the final section.

2 ADBASE and its basic parallel algorithm

Consider an MOLP formulated as follows:

$$max\{z = Cx \mid x \in S\}$$

where

$$S = \{x \geq 0 \mid A_{m_1} x \leq b_{m_1}, A_{m_2} x = b_{m_2}, A_{m_3} x \geq b_{m_3}\},$$

C is an $k \times n$ matrix, A_{m_i} are $m_i \times n$ matrices and b_{m_i} are $m_i \times 1$ vectors with nonnegative components, $i = 1, 2, 3$. The software ADBASE generates all efficient extreme points and unbounded efficient edges of MOLPs. A point x_0 in S is called an efficient solution of an MOLP if there is no other point x in S such that $Cx \geq Cx_0$, with strict inequality holding for at least one component.

In general, solving an MOLP can be viewed as finding a subset of all extreme points associated with the feasible set S, which is somehow similar to solving a linear program with multiple solutions. ADBASE consists of three main phases. In Phase 1, a single objective linear program (SOLP) related to the original MOLP is solved for an initial feasible extreme point of the MOLP. Phase 2 searches for an initial efficient extreme point (IEEP) of the problem, and Phase 3 includes generating all efficient extreme points and unbounded efficient edges. In Phase 3, all nonbasic variables of an IEEP are checked for feasibility and efficiency, which identifies all efficient extreme solutions adjacent to the initial one. The feasibility and efficiency test is continued at efficient extreme points subsequently found by performing simplex pivot operations between a current and adjacent efficient extreme point (this operation will be referred to as the *efficient pivot operation*). The bookkeeping includes storing EEPs that have been already found and their bases (referred to as *efficient bases*). The process goes on until all solutions are generated. Assigning the efficient solutions to nodes and efficient edges to arcs, one can construct a graph (referred to as the *EEP-graph*) along which the search can be performed. Since using this procedure, Phase 3 dominates computations and searching in the graph, the parallelization is naturally started from there.

The operation of moving from one coded basis to another is called *crashing* [12] and the related subroutine CRASH performs it in ADBASE. Once a processor finished working on a current basis, it may crash to another one by performing a required number of not necessarily feasible pivots that are needed to move between the two bases.

A basic parallel algorithm presented below is based on the assumption that each of processors searches a subgraph of the EEP-graph along the nodes and edges that are generated by itself with minimum overlapping, so that very limited bookkeeping has to be employed. A processor has its own

list, on which efficient bases are coded as '0' when a basis is found by itself, or as '1' when a basis is found by other processors.

Basic Parallel Algorithm:

- All processors find an identical IEEP by running Phase 1 and Phase 2 of ADBASE.
- Statically assign the nonbasic variables of this IEEP to all processors.
- In parallel, do on all processors:
  1. Examine current nonbasic variable.
  2. If a new efficient basis is found, broadcast a coded message, called "list message", to all the other processors and put it on its own list with code '0'.
  3. Check its buffer for possible new bases sent by other processors and update its list accordingly.
  4. If there is a subsequent efficient basis with code '0' on its list, crash to it and go back to Step 1; otherwise, go to Step 5.
  5. Send a "done message" to all the other processors since it has finished examining all efficient bases found by itself.
  6. Receive either "list message" or "done message" until all done messages from the other processors have been received.

This basic parallel algorithm has been tested on an Intel iPSC/2 hypercube machine. Table 1 shows the parallel execution times for p processors as well as for sequential ADBASE ($p = 1$) on 6 small testing problems. The second column of this table specifies the number of efficient bases and EEPs. For example, problem #1 has 20 efficient bases and 17 EEPs, while problem #3 has 21 efficient bases and 21 EEPs. Parallel execution time is determined by the slowest processor. In order to see the parallel efficiency of the algorithm, the shortest time used by a processor is listed inside parentheses. One can see that in all cases but one, this basic parallel algorithm did better than the sequential algorithm and more processors solved the problems faster.

The algorithm, however, has two disadvantages. Firstly, it suffers from severe job imbalance. Each processor examines only those efficient solutions that have been found by itself. The nonbasic variables, that lead to efficient

Table 1: Execution Time of the Basic Parallel Algorithm (Seconds)

Problem	No. of Solutions	$p = 1$	$p = 4$	$p = 8$
#1	20bas 17ex	.515	.312 (.003)	.227 (.002)
#2	25bas 16ex	.701	.373 (.006)	.346 (.002)
#3	21	.833	.505 (.006)	.412 (.002)
#4	20bas 14ex	.855	.614 (.006)	.450 (.007)
#5	46	4.829	1.975 (.021)	1.706 (.021)
#6	55	4.784	4.796 (.016)	2.400 (.016)

pivots, make their processors work and progress through the graph, while the processors assigned to the nonbasic variables that failed the efficiency test stop working and become idle even that there are still many efficient points to be found. Secondly, the algorithm is very much dependent upon the IEEP. An IEEP with more efficient pivots would allow more processors to do actual work and thus result in a faster performance. It was believed that the large time discrepancies between the slowest and the fastest processors shown in Table 1 are mainly caused by these two shortcomings of the algorithm.

3 Parallel algorithm

In this section, we shall discuss several improvements made on the basic parallel algorithm and present a more advanced algorithm.

The advantage of parallel computation can be easily lost when the load is unbalanced. Very visibly, the basic parallel algorithm described in Section 2 suffers from severe job imbalance. The nature of the MOLP makes the task of job balancing difficult. First, subgraphs of the EEP-graph have to be searched dynamically, since they cannot be equally distributed among processors before the execution. Second, re-activating idle processors unavoidably increases the bookkeeping complexity, communication, as well as redundant computations. In order to have each processor work until all the efficient points are found, the strategy, called *recrashing*, is proposed. The dynamic search of the subgraphs and recrashing are now described.

In the basic parallel algorithm, if a new basis has been found, a processor normally just sends the coded basis to all other processors. Now along with sending this, the processor also sends a number, referred to as "work

number", indicating its working status. For example, a '0' implies the processor is still working on a basis found by itself, '1' is a done message, and '-1' reports that this processor has re-started and is working on a basis sent to it. Each processor also has an integer Num_done on its list. This number indicates how many processors are not working anymore. The program terminates when all processors have stopped working. When a processor receives message from other processors, it adds attached work number to its Num_done. For instance, when '-1' is received by a processor, it knows that a previously idle processor has begun to work again and decrements its Num_done.

When a processor finished examining all bases found by itself and sent a done message '1' to all other processors, instead of simply receiving messages and waiting for other processors to finish their search, this processor will crash to any coming new basis and perform the efficient pivot operation on it. If a new efficient solution is found, its code together with the work number '-1' are sent out. The processor then progresses from there and searches the rest of the graph. Note, when a processor finds a new efficient solution it will not start searching from this solution until it has finished working on its current basis. Then the processor will crash to the next basis in its array and work from there. Thus, when an idle processor receives a new basis, it will most likely work on this new basis prior to the processor that sent the basis. It was conjectured that this distribution of the work should speed up the process of solving an MOLP.

As far as the sensitivity of the basic algorithm to the initial solution is concerned, a natural remedy is to give each processor a different initial point to work with, since there is no guideline for generating an IEEP with more efficient pivots. For this, Phase 2 must be able to robustly provide multiple initial efficient solutions. Two approaches were initially tried. The first one attempted was to use the random weight method (RANDWEIGHT) to find the initial solutions. In the random weight method the composite function is formed by randomly weighting objective functions of the original MOLP and this single objective function is maximized over the original feasible set. Relationships between MOLPs and the weighting method in general are discussed in detail in [5], [12], and many other related publications. Applying the random weight method in Phase 2 would then possibly lead to finding up to p different IEEPs for p processors used. However this was not the case. Among our testing problems, the most frequent number of IEEPs found was one and the maximum number found was three. The other approach was to have different processors use different methods employed in Phase

2. In general, five options for finding an IEEP are provided by ADBASE. Three of them involve lexicographic maximization and two involve the equal weight method. In our experiments, the equal weight method was assigned to half of the processors and the lexicographic method to the other half (WEIGHT&LEX). A comparison of RANDWEIGHT and WEIGHT&LEX in Phase 2 using 8 processors was conducted. It suggested that the latter worked better in general, because in all cases it found two different initial solutions.

The strategies discussed above, i.e., the technique of recrashing for activating idle processors and the combination of the equal weight and the lexicographic method for generating different IEEPs, have been tested on an Intel iPSC/2 machine. Results on the same testing problems as in Table 1 are listed in Table 2 that includes the execution time (in seconds) and the *speedup*, defined as

$$\text{speedup} := \frac{\text{Execution time using 1 processor}}{\text{Execution time using } p \text{ processors}}. \tag{1}$$

The speedups of the basic parallel algorithm on the same problems are listed inside the parentheses for comparison. Table 2 clearly shows that these two strategies have significantly increased the efficiency of the basic parallel algorithm in almost all cases even though this new version of the algorithm may involve more redundant computations.

Table 2: Testing Results of Proposed Strategies

	$p = 4$		$p = 8$	
Problem	Time	Speedup	Time	Speedup
#1	.182	2.83 (1.65)	.194	2.65 (2.27)
#2	.483	1.45 (1.88)	.418	1.68 (2.03)
#3	.372	2.24 (1.65)	.353	2.36 (2.02)
#4	.336	2.54 (1.39)	.232	3.69 (1.90)
#5	1.904	2.54 (2.45)	1.077	4.48 (2.83)
#6	1.776	2.69 (1.00)	1.022	4.68 (1.99)

The use of the two available subroutines in ADBASE, the random weight method and lexicographic method, for generating different IEEPs was primarily for the convenience of initial testing, and certainly should not be

recommended for a parallel algorithm that allows concurrent execution on large number of processors. In fact, the approach of employing different methods for solving the same initial SOLP in order to simultaneously generate multiple IEEPs is impractical. The theory of multicriteria optimization does not address the issue of multiple IEEPs. The number of existing methods for finding an IEEP is far less than the number of processors available. In addition, applying different methods concurrently on multiprocessors would result in programming complexity and load imbalance, which obviously prohibits practical usage of such an approach.

After a careful study of the methods used by ADBASE, it was found that formulating multiple SOLPs from the given MOLP and solving these SOLPs by the same method would be a better approach. Actually, a small modification of ADBASE was quite satisfactory. ADBASE is capable of performing the lexicographic maximization process that is carried out in accordance with the recursively defined reduced feasible regions:

$$\begin{aligned} S_0 &= S \\ S_1 &= \{y : c^1 y = max[c^1 x \mid x \in S_0]\} \\ &\vdots \\ S_i &= \{y : c^i y = max[c^i x \mid x \in S_{i-1}]\} \\ &\vdots \\ S_k &= \{y : c^k y = max[c^k x \mid x \in S_{k-1}]\}, \end{aligned}$$

with

$$C = \begin{bmatrix} c^1 \\ \vdots \\ c^k \end{bmatrix}.$$

In particular, the process maximizes the objective functions in the order in which they are stored by rows in criterion matrix C. Obviously, a different maximization process can be obtained from a reordering of the objective functions, or equivalently, a reordering of rows in matrix C. There are $k!$ orderings in all, a number usually much larger than the number of processors available.

A short subroutine that permutes the rows of C was then added into ADBASE. Experiments on all orderings for MOLPs with k objective functions

show that the orderings:

$$\begin{array}{rcl} S_0 & = & S \\ S_1 & = & \{y : c^i y = max[c^i x \mid x \in S_0]\} \\ S_2 & = & \{ \qquad\qquad \cdots \qquad\qquad \} \\ & \vdots & \\ & , & \end{array}$$

$i = 1, \cdots, k$, are guaranteed to generate k different IEEPs. That is, different sets S_1 in this process are guaranteed to produce different IEEPs. When the solution in S_1 is unique, changing the orderings of objective functions in subsequent sets S_i, $i = 2, \cdots, k$, makes no difference in IEEPs produced, which has occurred in the test.

Incorporating the two techniques: recrashing and producing multiple IEEPs by means of the lexicographic process resulted in the final parallel algorithm. Step 1 of the algorithm below refers to an SOLP related to the MOLP being solved. This SOLP is originally formulated in Phase 2 of ADBASE, now equipped with the additional subroutine permuting the rows of C.

Parallel Algorithm for MOLPs:

- In parallel, do on each of processors P_i, $i = 1, \cdots, p$, until $Num\_done = p$:

  1. Formulate an SOLP from the given MOLP. Find an IEEP by solving this SOLP.
     Initialize $Num\_done := 0$.
  2. Follow Steps 1-5 of the Basic Parallel Algorithm.
  3. Receive messages from other processors and update its own list.
  4. When a new coded basis is received:
     Crash to and do efficient pivot operation on it.
     If the basis leads to a new efficient solution:
     send its code with the work number '-1' to all other processors;
     go back to Step 2.
     Otherwise, go to Step 3.

This parallel algorithm has been implemented on the Intel Paragon multiprocessor at NASA Langley Research Center. Experimental results are given in the next section.

4 Numerical results

Parallel algorithms are developed for solving computationally extensive problems. The speedup, efficiency and scalability are important criteria in the performance evaluation. The scalability referred to in this paper is understood as the following feature of the algorithm: when the problem size increases linearly with the number of processors, the achieved *efficiency* of the algorithm, defined as

$$\text{efficiency} := \frac{\text{speedup on } p \text{ processors}}{p}, \tag{2}$$

is maintained. Clearly, the testing problems should consist of MOLPs whose size is scaled with the number of processors. Fortunately, using ADBASE, almost any problem of desired size can be generated by specifying the number of objective functions (k), number of structural variables (n), and number of constraints (m_1).

An interesting phenomenon initially observed is that the number of EEPs grows rapidly as $k+n+m_1$, sum of the parameters, increases. For example, the solution sets could grow by several thousand when the number of objective functions is incremented by one. Similar observations were reported in [14], where random problem generation for creating MOLP test problems was discussed.

In general, an MOLP may have a huge number of solutions, which is beyond capability of data processing or exceeds the machine memory capacity. In this situation, finding all efficient solutions is no longer practical and the goal of generating all EEPs could be questioned. However, in the presence of a large number of EEPs, the solution process of MOLPs goes further and involves maximizing a decision maker's overall utility function over the efficient set in order to obtain a (possibly unique) most preferred solution. Optimization over the efficient set has recently become a direction of very active research. In fact, optimizing a linear function over the efficient set of an MOLP is already a difficult global optimization problem and requires numerically intensive algorithms. Studies in this direction, yielding exact or heuristic algorithms, were carried out by Benson [1] [2], Benson and Sayin [3], and others. While exact algorithms entail heavy computational burden, heuristics offer only estimates of the global solution. Given the availability of fast parallel algorithms for MOLPs, complete enumeration of EEPs, previously considered impractical for larger problems, seems to be competitive with the specially designed algorithms. The global solution obviously

Table 3: Execution Time (Seconds)

Problem	No. of Solutions	$p = 1$	$p = 2$	$p = 4$	$p = 8$
#7	58	0.7724	0.4951	0.3573	0.3852
#8	129	1.9827	1.0944	0.6551	0.6988
#9	251bas 234ex	5.0659	2.7440	1.8469	1.0997
#10	424bas 402ex	13.4987	7.3755	5.1234	2.7138
#11	818	32.6500	17.0657	9.4557	6.1187
#12	1512bas 1473ex	89.7754	46.9689	25.3096	17.9618
#13	3119	146.1361	76.0051	43.0819	23.7395

depends on the choice of the utility function and as such it cannot be determined uniquely. Selection of the utility function is a difficult task usually performed by a decision maker, who can make a better choice if more information about the efficient set is available. Therefore, ability to find a subset of the efficient set within a given time period is of great importance, as it may contribute to the decision maker's learning process about the efficient frontier. Using that partial information, the decision maker can modify the utility function that will better represent his/her preferences.

In this paper, MOLPs that involve more than 5 thousand EEPs are classified as extremely large problems. Accordingly, for small and large MOLPs, the goal is to generate all EEPs as fast as possible; otherwise, to find as many solutions as possible within a given time period.

Table 3, constructed in the same fashion as Table 1, shows the parallel execution times on the Intel Paragon machine on 7 testing problems. The problems are selected so that their sizes, measured by the total number of efficient bases, are linearly scaled with the number of processors being used. Table 4 lists the speedup for the same testing problems and Figure 1 depicts the parallel efficiency defined by Eq.(2) for all the problems tested in the experiments.

Figure 1: Parallel Efficiency

Table 4: Speedup

Problem	No. of Solutions	$p = 2$	$p = 4$	$p = 8$
#7	58	1.56	2.16	2.01
#8	129	1.81	3.03	2.84
#9	251bas 234ex	1.85	2.74	4.61
#10	424bas 402ex	1.83	2.63	4.97
#11	818	1.91	3.45	5.34
#12	1512bas 1473ex	1.91	3.55	5.00
#13	3119	1.92	3.39	6.16

The structure of the EEP-graph suggests that, when an MOLP includes more solutions to be found, the chances of splitting the work between processors will increase, because there is less chance for more than one processor to be terminated at the same time and thus, to recrash to the same solution. Therefore the algorithm is inherently scalable, which has been confirmed by the experiments. Note the three stars marked in Figure 1. They indicate that the efficiency of the algorithm has been maintained when the total number of solutions generated increased linearly with the number of processors employed. In addition, for a fixed number of processors, the efficiency of the parallel algorithm went up quickly to its optimum when the number of solutions of MOLPs increased, as illustrated in Figure 1.

The implementation results on extremely large MOLPs are presented in Table 5. These problems involve huge numbers of solutions, so they are identified by the input parameters used in the problem generator of ADBASE. Three groups of testing problems were chosen. The problems in each group are defined over the same feasible set, specified by (n, m_1), and are listed in the ascending order according to their number of objective functions. Solutions were found in 30 seconds using p processors. All the rows of this table show that, in general, more solutions were generated simultaneously when more processors were used. Since the speedup defined by Eq.(1) is based on the execution time spent on generating all efficient solutions and is no longer valid for the performance evaluation in this situation, the *ratio*, defined as

$$\text{ratio} := \frac{\text{number of solutions found by } p \text{ processors}}{\text{number of solutions found by 1 processor}}, \tag{3}$$

is then used. The ratio actually measures the speedup of the computation

Table 5: Number of Solutions Found (in 30 Seconds)

(n, m_1)	k	$p=1$	$p=2$	$p=4$	$p=8$	$p=16$	$p=32$
(100,30)	16	120	142	179	264	439	499
	24	49	83	127	221	369	535
	32	23	40	82	194	332	628
(80,67)	16	87	133	264	334	510	564
	24	61	73	126	196	406	598
	32	32	74	101	180	348	671
(150,50)	16	18	20	33	35	35	35
	24	9	12	32	34	34	44
	32	5	13	24	27	27	42

process in terms of the number of solutions found within a given time period. Table 6 lists the ratios computed from the data in Table 5. For the given (n, m_1) as parameters of a feasible set, the ratios in most of the columns of Table 6 increase steadily with the number of objective functions, indicating once again that the algorithm generally performs better on larger problems and is well scalable. Note that the number of solutions generated by 32 processors can be as large as 27 times of the number of solutions found by 1 processor in the same time period.

5 Conclusions

This paper presents pioneering research on designing a parallel algorithm for MOLPs based on the sequential software ADBASE and implementing it on an Intel iPSC/2 and a Paragon multiprocessors. The paper first reports a straightforward approach in the form of basic parallel algorithm. In the subsequent research, the techniques of re-activating idle processors and generating multiple IEEPs have been proposed. The resulting parallel algorithm has been applied to large and very large MOLPs. All experiments show that this parallel algorithm significantly speeds up the process of finding efficient solutions of MOLPs. Furthermore, the algorithm is well scalable, which is considered a very important feature of parallel algorithms. The algorithm is also very well suited for a wide range of parallel computers and it is not specific to the distributed multiprocessors on which it was tested.

Table 6: Ratio

(n, m_1)	k	$p = 2$	$p = 4$	$p = 8$	$p = 16$	$p = 32$
(100,30)	16	1.18	1.49	2.20	3.66	4.16
	24	1.69	2.59	4.51	7.53	10.92
	32	1.74	3.57	8.43	14.43	27.30
(80,67)	16	1.53	3.03	3.84	5.86	6.48
	24	1.20	2.07	3.21	6.66	9.80
	32	2.31	3.16	5.63	12.00	20.97
(150,50)	16	1.11	1.83	1.94	1.94	1.94
	24	1.33	3.56	3.78	3.78	4.89
	32	2.60	4.80	5.40	5.40	8.40

Although the literature on MOLPs is very rich and diverse, computational issues of these problems have not been widely investigated. A secondary product of this research is the report on the numbers of EEPs possessed by MOLPs of large and very large sizes as well as on mutual relationships between the number of objective functions, variables, and constraints.

Additionally, the current structure of ADBASE heavily affects this algorithm and leaves space for further improvement. For instance, since ADBASE does not keep track of infeasible or inefficient bases, currently in the parallel algorithm multiple processors repeatedly check the same infeasible or inefficient bases, which generates redundant computations. If bookkeeping of inefficient bases was maintained, the communication between processors could be set up for transmitting the additional information about efficient and inefficient bases. On the other hand, ADBASE is a versatile package that can solve a range of linear optimization problems, i.e. pre-emptive goal programming, MOLPs with interval criterion weights, and point estimate weighted-sum problems. The current parallel algorithm could be further modified and extended to handle some of the other options ADBASE offers.

The ultimate goal of any research in the area of multicriteria optimization is to design new tools supporting decision making. In the course of this process, the decision maker usually interactively examines the efficient set and chooses a most preferred efficient solution as the optimal one. Optimizing decision maker's preferences over the efficient set, although in general considered a difficult problem, has been more attractive than generating all

efficient points by means of traditional sequential algorithms. The research presented in this paper shows that parallel algorithms can substantially alleviate this tedious process and make enumeration of efficient points a decision aid for multicriteria decision making.

References

[1] H. P. BENSON, *An all-linear programming relaxation algorithm for optimizing over the efficient set*, Journal of Global Optimization, 1 (1991), pp. 83–104.

[2] ——, *A finite, nonadjacent extreme-point search algorithm for optimization over the efficient set*, Journal of Optimization Theory and Applications, 73 (1992), pp. 47–64.

[3] H. P. BENSON AND S. SAYIN, *A face search heuristic algorithm for optimizing over the efficient set*, Naval Research Logistics, 40 (1993), pp. 103–116.

[4] C. S. CHANG, *Co-ordinated static and dynamic monitoring and optimization of power systems using a parallel architecture and pattern recognition techniques*, IEEE Proceedings - C, 139 (1992), pp. 197–204.

[5] V. CHANKONG AND Y. Y. HAIMES, *Multiobjective Decision Making - Theory and Methodology*, North-Holland, New York, 1983.

[6] J. N. CLIMACO, J. P. COSTA, C. ANTUNES, AND M. J. ALVES, *Parallel processing in molp method base development - discussion using two case studies*, Paper presented at the IX-th International Conference on Multiple Criteria Decision Making, Fairfax, VA, (1990).

[7] J. P. COSTA AND J. N. CLIMACO, *A multiple reference point parallel approach in mcdm*, Proceedings of the Tenth International Conference on Multiple Criteria Decision Making, Taipei, 3 (1992), pp. 265–272.

[8] Y. EVTUSHENKO, V. MAZOURIK, AND V. RATKIN, *Multicriteria optimization in the diso system*, Optimization, Parallel Processing and Application, eds: A. Kurzhanski, K. Neumann and D. Pallaschke, Springer-Verlag, Berlin, (1988), pp. 94–102.

[9] M. Grauer and H. Boden, *Opportunities on parallel and distributed computation for optimization and decision support*, Proceedings of the Tenth International Conference on Multiple Criteria Decision Making, Taipei, 1 (1992), pp. 197–207.

[10] A. Lewandowski, *Parallel implementation of selected mcdm algorithms*, Paper presented at the TIMS/ORSA Joint National Meeting, Chicago, (1993).

[11] W. Y. Ng and J. Yang, *Interactive sampling of efficient frontier in multi - objective programming by parallel distributed computation*, Proceedings of the Tenth International Conference on Multiple Criteria Decision Making, Taipei, 2 (1992), pp. 325–334.

[12] R. E. Steuer, *Multiple Criteria Optimization - Theory, Computation and Application*, John Wiley, New York, 1986.

[13] ——, *Manual for the adbase multiple objective linear programming package*, Department of Management Science and Information Technology, University of Georgia, Athens, GA, (1991).

[14] ——, *Random problem generation and the computation of efficient extreme points in multiple objective linear programming*, private communication, (1993).

[15] M. M. Wiecek, H. Zhang, J. L. Matthews, and J. R. Soltys, *A parallel algorithm for multiple objective linear programs*, to appear in Proceedings of the XI International Conference on MCDM, Coimbra, Portugal, (1994).

[16] H. Zhang and M. M. Wiecek, *Solving multiple objective linear programs on the intel paragon*, to appear in Proceedings of Mardi Gras '94 Conference: Toward Teraflop Computing and New Grand Challenge Applications, Baton Rouge, Louisiana, (1994).